Inhalt

[1. General notes: 2](#_Toc95108600)

[1.1 Inline functions: 2](#_Toc95108601)

[2.1 Header files: 2](#_Toc95108602)

[3.1 2](#_Toc95108603)

[2. Memory: 2](#_Toc95108604)

[4.1 Cache allocator 3](#_Toc95108605)

[3. Kernel: 3](#_Toc95108606)

[4. Kernel Mode VS User Mode: 3](#_Toc95108607)

[5. Process: 3](#_Toc95108608)

[5.1 Process termination: 3](#_Toc95108609)

[6.1 Process state: 3](#_Toc95108610)

[7.1 task\_struct: 4](#_Toc95108611)

[8.1 Init process: 4](#_Toc95108612)

[9.1 Daemon process: 4](#_Toc95108613)

[6. Scheduler: 4](#_Toc95108614)

[7. System call: 5](#_Toc95108615)

[8. I/O on file: 5](#_Toc95108616)

[10.1 Open system call: 5](#_Toc95108617)

[9. File descriptor: 5](#_Toc95108618)

[10. Process: 5](#_Toc95108619)

[11. Threads: 6](#_Toc95108620)

[11.1 Mutex: 6](#_Toc95108621)

[12.1 condition variables: 6](#_Toc95108622)

[13.1 Thread Cancellation: 6](#_Toc95108623)

[12. Data structure: 6](#_Toc95108624)

[14.1 Linked list: 6](#_Toc95108625)

[15.1 Koject: 7](#_Toc95108626)

[13. Interrupts: 7](#_Toc95108627)

[16.1 Bottom half: 7](#_Toc95108628)

[Shared library: 8](#_Toc95108629)

[14. Time: 8](#_Toc95108630)

[17.1 RTC: 8](#_Toc95108631)

[18.1 Real Time: 8](#_Toc95108632)

[19.1 system timer: 8](#_Toc95108633)

[20.1 Timers 8](#_Toc95108634)

[21.1 Delay 8](#_Toc95108635)

[15. Signals: 9](#_Toc95108636)

[16. VFS: 9](#_Toc95108637)

[17. Virtual address space: 9](#_Toc95108638)

[18. IPC: 10](#_Toc95108639)

[22.1 Critical sections: 10](#_Toc95108640)

[23.1 Semaphors: 10](#_Toc95108641)

[24.1 Atomic operations: 10](#_Toc95108642)

[25.1 Spinlocks: 10](#_Toc95108643)

1. General notes:

UMA: unified memory access -> all processors can access all memory

NUMA: each processor has a memory which he can access fast and can access another memory which is local for the other processor but slower

Void \* = unsigned long -> as void is architecture dependent Linux uses long to make sure that the code will be the same regardless of the architecture

1. Inline functions:

Normally it is written inside header file and then included in the file which is going to use it as u must define the function before u can use it

1. Header files:

* It is inside include directory in Linux source tree

2. Memory:

Kernel divide physical memory to Nodes -> zones -> pages

Zones are -> DMA, normal and high memory

Pages is the virtual memory unit while frame is the physical memory unit

The Virtual Memory is divided 3:1 -> lower 3 GiB is for user mode while the upper 1 Gib is for Kernel mode

Fourth GiB address space is for the kernel starts from, \_\_page\_OFFSET

There is two micros to convert between physical and virtual address in kernel land

* + \_\_pa() -> would return physical address
  + \_\_va()-> would return virtual address

Allocation of memory is done using alloc\_pages() -> in returns physical address

To convert the physical to logical address page\_address()

Get\_free\_pages() -> will return logical address

Kmalloc() -> allocate memory which are continuous physically and virtually

Vmalloc() -> allocate memory which is continuous virtually

1. Cache allocator

* Cache stores different type of object
* Slab is one or more page
* Slab contain object
* Cache is full, empty, or partially and contains slab

1. Page table:

* It is used to create a relationship between virtual address space and the physical address space
* Addresses in virtual memory is divided in 5 parts, 4 for the table and 1 for the offset
* BITS\_PER\_LONG -> define how many bits it is used to define long data type

1. Kernel:

Resource Manager

Scheduling

Memory management

Provide system API

Provide file system

It has no access to C library or standard header file

<linux/notify.h> -> would be in include/linux/notify.h

1. Kernel Mode VS User Mode:

Kernel mode can access the data in user and in kernel land

User mode can access the data only in user land

1. Process:

Instance of executing program

Kernel identify the process by PID

Kernel schedule thread not the process

Kernel loads it in the memory and provide resources for it

Process can create another program by calling fork()

Kernel will duplicate text, data, stack, heap of the parent to the child

The child calls execve() to replace the old data, stack heap, text

1. Process termination:

* Process can be terminated by calling exit or be receiving killing signal
* The parent can get the exit status of the process by calling wait()

1. Process state:

* It could be running, interruple, stopped

1. task\_struct:

* it has all the information the kernel needs to work with the process
* it is double linked list
* it represents a process
* it has thread\_info structure
* it has the location of the stack
* it has element children which is a list head to the childrent structure which u can use to go through the whole process

1. Init process:

* it has the ID 1
* it cannot be killed

1. Daemon process:

* it runs in background
* long live process

1. Scheduler:

Process can be I/O or processor bound

The idea is that the high priority should run first and longer than the process with less priority

Priority is nice from -20 to 19 and real priority from 0 to 99

CFS: complete fair scheduler

Time slice: it is the amount of the time the process is going to given to run

CFS: gives a slice of the time slice to each process according to nice value

The problem with scheduler:

* + If u map the priority to the time slice then u want get a good result as the time different between two process which there value are 0 and 1 wont be the same as the different between two processors which have the value 18 and 19

Preemptive concept:

* Process can preemptive another process in user land
* If the kernel is in system call no process can preemtive it rather than an interrupt

1. System call:

It an entry point from user land to kernel land

Application uses c library functions to execute system call

The c library will use int machine instructions to makes the CPU jump to a specific place in kernel land which will have routine to handle system call and will update s register in cpu which has system call number, if error happens y library will update errno to say that error has happened

If the system call failed it will return -1 and update errno variable, u should check only the errno if the system call returned -1 as a successful system call won’t reset errno

1. I/O on file:

File descriptor referres to all opened files (PIPE, FIFO, SOCKET)

Any process has 3 standard file descriptors

* + 1 -> stdin
  + 2 -> stdout
  + 3 -> stderror

To change the standard file descriptor use fropen()

1. Open system call:

* Flag will be access mode and to create the file or truncate it
* Mode will be the permission in this file which makes sense only if u create the file for the first time

1. File descriptor:

There is not one to one match between file descriptor and open file

Using Open, dup, or fork will create new file descriptor which will refer to the same global file descriptor which will refer to the same i\_node

An example of dup is redirect of error message 2>&1 -> dup2(2,1)

1. Process:

Every process has process id which u can get by calling getpid(void)

To get the parent of the process u can call getppid(void)

Kernel divide physical RAM into frame ad Virtual Ram into Pages

Page -> VM / Frame -> Physical

There is a page table between page and frame for each process which is maintained by the kernel

Process is created by using fork() which create a process as a copy of the parent process

Execv() will replace the process with a new process

U can assign handler to be called on exit()

System() -> allow writing shell in C script

1. Threads:

Every thread has errno variable to prevent race condition

Pthread\_t -> datatype for a thread

Pthread\_create() -> will create a thread and execute it

Pthread\_exit() -> can be called by any functions which is called from pthread\_create

Pthread\_join() -> wait on a specific thread to be joined cancelled

Pthread\_detach() -> to make the thread not joinable to save resources in the system

If the main exit() all the process are cancelled

1. Mutex:

* It can be statically or dynamically created
* static pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;
* s = pthread\_mutex\_lock(&mtx)
* or pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr);
* int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);
* attribute:
  + PTHREAD\_MUTEX\_RECURSIVE -> count. When a thread first acquires the mutex, the lock count is set to 1. Each subsequent lock operation by the same thread increments the lock count
  + s = pthread\_mutexattr\_settype(&mtxAttr, PTHREAD\_MUTEX\_ERRORCHECK);

1. condition variables:
2. Thread Cancellation:

* Pthread\_cancel(pthread\_t thread) -> to cancel a thread
* Pthread\_cancelstate() -> u can make the thread not cancellable, or u can assign the thread that it only cancel when it reaches a specific point

1. Data structure:
2. Linked list:

* The first element is called head
* The list\_head structure is embedded in every element in the list with the name list
* List\_head it has prev and next to point to the next and the last element we need to point to
* List\_entry() -> use container of to get the struct which has the embedded element
* To initialize the element in the list we use 2 possible ways:
* INIT\_LIST\_HEAD() -> dynamic
* LIST\_HEAD\_INIT() -> static
* To create the head : LIST\_HEAD(name\_of \_head)
* List\_add() -> add after the head
* List\_del() -> to delete element from the list
* To iterate over the whole list and get the struct use list\_for\_each\_entry(whole struct, struct head, name\_of\_head)
* List\_for\_each\_entry\_safe() -> u can delete the element and keep iterating over the list

1. Koject:

* The role of kobject ist:
  + Providing reference count
  + Locking sets
  + Export properties using sysfs
  + Managing lists of objects
  + Kobject is embedded in the structure

1. Interrupts:

Interrupt enables the hardware to signal the operation system that something has happened

Interrupt controller: device which multiplex interrupt lines to a single line in the processor

IRQ: interrupt request which is an integer number to distinguish between different sources of interrupt

ISR: interrupt service routine which will run when the interrupt is fired

Interrupt handler runs in interrupt context without process which he is related to so it cannot sleep

Interrupt handle while other interrupts are disabled that is why it has to be really fast to avoid the situation where the interrupt might be lost

To register a handler to a specific interrupt request\_irq()

* + If u used shared interrupt u have to path a unique value to dev so the kernel knows which one to remove if u decided to free the interrupt line

1. Bottom half:

* It runs while other interrupt as enabled
* Softirq:
  + It is an array of pointers of functions which will be called later

1. Shared library:

Place a function in single place which can be used by multiple process at the same time

1. Time:

Tick: smallest time unit in the system

HZ: how many ticks in one second

HZ affects the timer interrupt which affects the scheduler

Jiffies: number of tickles since booting

1. RTC:

* Real time clock
* Non volatile device which save system time
* It keeps tracking of the time even if the power is off
* On boot the kernel reads RTC and use it to update wall time

1. Real Time:

* It is calendar time or wall clock time (time elapsed since a specific point)
* gettimeofday(timeval) -> assign the seconds and millisecond since EPOC in time\_t
* time(NULL) -> will return seconds since EPOC
* ctime(time\_t) -> return time as string wed Jun 8 14:22
* struct tm \* localtime(time\_t) -> get the time to broken down time year, month
* mktime() -> convert broken down time to EPOC time
* asctime() -> convert to wed Jun 8 14:22

1. system timer:

* it is used to provide a mechanism for driving the interrupt at periodic time
* PIT: programmable interrupt time
* Kernel program PIT to interrupt the processor at HZ frequency
* Every interrupt the kernel will update the wall time and fire the expired timers

1. Timers

* It is used to delay execution of some function until a later time in the future
* Struct timer\_list represent timer
* Init\_timer(&) -> to initialize the timer
* Assign the values after initialization the timer
* Add\_timer(&timer) -> to fire the timer so the timer starts counting
* del\_timer() -> the job is to prevent the timer to be fired in the future

1. Delay

* It is way to give the hardware time
* While (time\_before(jiffies, jiffies + 5));
* Udelay() -> it is used to delay for time without using jiffies as the processor is faster than jiffies the function is implemented depending on the time the processor is using to execute an instruction
* U can schedule the task to sleep using:
  + Set\_current\_state() , schedule\_timeout()

1. Signals:

Software interrupt

Process can deal with the signal in three different ways: Ignore, default or handler

Famous signals are:

* + SIGTERM: it is used to stop the process
  + SIGKILL: kill the signal u cannot assign handler to it so it makes sure that u end the process it is equal to KILL -9
  + U can assign a handler to a signal using signal() functions
  + U can send signal to another process using KILL(pid, signal) command

1. VFS:

It enables different file system to exist and cooperate with each other

It represents files and data

It allows different system call to work regardless of which file system is available

Virtual file system makes linux support different physical file system without a problem for example user space write would call sys\_write”which is VFS” which would call write in the physical media

Linux organize file system using 4 concepts

* + File: which is collection of bytes
  + Folder: which is file which has information about the files
  + Inode: it has the information about the file such as size, permission, etc, it is stored as superblock which has information about the file system as a whole

1. Proc:

* It is on the fly file system
* It is used to communicate with the kernel
* It is used for:
  + Memory management
  + File system
  + Device drivers
  + System bus
* The main purpose of Proc is to deliver process data
* Every entry is an instance of proc\_dir\_entry

1. Sys:

* It is on the fly file system

1. Virtual address space:

Virtual address space of a process has sections:

* + Text segment where there is a code
  + Stack
  + Heap
  + Data

Address space of the memory is done in the elf file

Memory mapping:

1. IPC:
2. Critical sections:

* Means only one process may enter this section at a given time

1. Semaphors:

* Number which can be positive or negative
* Two operations are available up and down
* The process call down if the variable is zero it can enter the section
* If the other processor called down it will sleep
* Down step is atom step
* Kernel puts the processor on sleep
* It works well in user land

1. Atomic operations:

* Atomic\_t -> it is the data structure to represent atomic variable

1. Spinlocks:

* It wont sleep if the lock is taken

1. Device Drivers:

I/O device:

* + Hardware must be addressed
  + User application must have tool to access the device
  + User space should know which device is available
  + I/O memory: port address of the I/O port is mapped into normal memory, kernel provided abstraction layer to map I/O to memory. CPU would go the I/O memory when u use this address
  + Each char device is represented by instance struct cdev
  + Kernel keeps all the character devices in array which index is the major number and has function which can return cdev struct
  + Cdev:
    - It represents char device
    - It has kobject, file\_operation, count

1. Resources management:

* All located io\_mem is children of iomem\_resource root

1. Bus system:

* Struct device is a way to represent device in kernel
* Struct device\_driver to represent a driver
* Struct bus\_type

1. System calls: